A Non-Blocking Concurrent Queue Algorithm

Bruno Didot
bruno.didot@epfl.ch
June 2012

Abstract
This report presents a new non-blocking concurrent FIFO queue backed by an unrolled linked list. Enqueue and dequeue operations can be run concurrently, without misbehaving.

Contents
1 Introduction 1
2 Algorithm 2
3 Correctness 3
   3.1 Safety .................................................. 5
   3.2 Linearizability ........................................... 7
   3.3 Lock-freedom ............................................ 8

1 Introduction
Many applications take advantage of multi-core computer architectures by executing multiple tasks concurrently, often accessing shared data concurrently. Correctness of these data structures is ensured by synchronizing concurrent accesses. There are generally two approaches to synchronization; mutual exclusion locks and lock-free algorithms. Locks are more traditional but do not come without any issue. On asynchronous, concurrently programmed, multi-core systems, a thread holding a lock might get delayed (e.g. by getting preempted by the scheduler or because of a page fault) or halted, making other threads wait until the lock is freed, thus preventing any progress to be made by those other threads. Similarly, if a slow thread gets the lock first, other faster threads will have to wait indefinitely for the lock to be released. More importantly, a failure in a thread holding a lock might lead other threads to wait indefinitely. In case of such events, lock-free algorithms are more robust.

A lock-free concurrent data structure guarantees that if several threads are trying to perform operations on a shared data structure, some operation will
complete within a finite number of steps. As such, operations on lock-free data structures are immune to deadlocks, do not get delayed by slower threads nor by preemption on other threads.

Many implementations of lock-free concurrent FIFO queues have already been proposed (see Hwang and Briggs, Sites, Michael and Scott). All of them rely on a linked list to achieve their goal. An unrolled linked list is slightly different from a standard linked list as each node stores multiple elements. This increases cache performance and decreases the memory overhead associated with storing a reference to the next node in the list for each node of the list and allocating a new node for each element in the list. Therefore, we hereby propose a FIFO queue based on an singly linked unrolled list.

2 Algorithm

The algorithm implements the queue as an unrolled singly linked list with head and tail pointers. head always points to a dummy node, which is the first node in the list. tail points to a node in the list. Tail is used to find the reference to the last node in the list without having to go through the entire list for each enqueue operation. Unlike a standard linked list, whose nodes store a single element, each node of an unrolled list stores multiple elements using an array.

The usage of a dummy node at the beginning of the list has first been proposed by Valoi, and allows easy and consistent maintenance of head and tail pointers. Using this node, we can guarantee that tail will never fall behind head; it will never point to a node that is not accessible through the head pointer. It also guarantees that both head and tail pointer will never be set to null.

In this implementation, elements are only enqueued at the first null slot in the unrolled list (the DELETED flag being non null). “First” null slot is intuitively defined as the array slot with the lowest index whose value is null, provided that all elements array slots in all nodes before the considered node are non null. A dequeue operation will only delete the first non DELETED element reachable from head.next. The special DELETED flag, which has first been proposed by Aleksandar Prokopec, has been introduced to distinguish two different states. Without the DELETED flag, node elements can only be deleted by being set to null. Therefore, operations on a queue can not distinguish an array slot that is null because it has never been set to any other value, with an array slot that is null because it previously held an element that has been deleted. This would result in an inconsistency with the enqueue definition.

An hint has been introduced for both enqueue and dequeue operations. Since elements are only deleted from the beginning of an element array, the position of newly deleted elements in the array is always increasing. Therefore, deleteHint has been added to prevent the loop that looks for the next element to be dequeued from checking all the slots in the array. deleteHint is then increased by the thread that has just finished dequeuing an element. The variable addHint behaves similarly.
Pseudocode 1 Data structures and initialization

head: Node
tail: Node
NODE_SIZE = integer constant
DELETED = flag constant

structure Node {
    next: Node
    elements: Array[NODE_SIZE]
    addHint: Int
    deleteHint: Int

    def Node(elem) =
        elements[0] = elem
        addHint = 1
}

def init() = {
    head = new Node()
    head.elements[0 until NODE_SIZE] = DELETED
    tail = head
}

3 Correctness

Definition 1 (Basics). An unrolled node (from now on node) is a structure holding a reference next to another node and an array of references elements to its elements.

Definition 2 (Unrolled queue). An unrolled queue is defined as the reference head to the root of the list. An unrolled queue state $S$ is defined as the sequence of nodes reachable from head. Nodes that are not reachable anymore are considered as deleted. The set of reachable nodes from a certain node can be expressed as:

$$reachables(node) = \begin{cases} 
\{node\} \cup reachables(node.next) & \text{if } node \neq \text{null} \\
\emptyset & \text{otherwise} 
\end{cases}$$

The values held by the array inside each consecutive node of a sequence of nodes can be concatenated into a string as:

$$elems(node) = \begin{cases} 
node.elements \cdot elems(node.next) & \text{if } node \neq \text{null} \\
\varepsilon & \text{otherwise} 
\end{cases}$$
We also introduce a relation to strip down leading `DELETED` and trailing `null` values of a string:

\[ \text{core}(s : String) = E^e \text{ if } s = DELETED^d \cdot E^e \cdot \text{null}^n, \text{ where } d, e, n \geq 0 \]

**Definition 3.** We define the following invariants for the unrolled queue.

- INV 1 `head \neq null`
- INV 2 `null \notin \text{head.elements}`
- INV 3 \(\text{tail} \in \text{reachables(head)}\)
- INV 4 `node \notin \text{reachables(node.next)}`
- INV 5 `\text{elems(head.next)} = DELETED^d \cdot E^e \cdot \text{null}^n` where \(d, e, n \geq 0, E \in \text{set of enqueued elements}\)
**Definition 4** (Validity). A state $S$ is valid if and only if the invariants INV 1-5 are true in the state $S$.

**Definition 5** (Abstract queue). An abstract queue $Q$ is a string of elements $e_0 \cdot e_1 \cdot \ldots \cdot e_n$, where $n \geq 0$. An empty abstract queue is defined by the empty string $\varepsilon$. Abstract queue operations are $enqueue(Q, e) = Q \cdot e$ and $dequeue(Q) = Q_1 : Q_1 = e_1 \cdot e_2 \cdot \ldots \cdot e_n$ if $Q = e_0 \cdot e_1 \cdot \ldots \cdot e_n$. Operations $enqueue$ and $dequeue$ are destructive.

**Definition 6** (Consistency). An unrolled queue state $S$ is consistent with an abstract queue $Q$ if and only if $Q = core(elems(head.next))$. A destructive unrolled queue operation $op$ is consistent with the corresponding abstract queue operation $op'$ if and only if applying $op$ to a state $S$ consistent with $Q$ changes the queue into $S'$ consistent with an abstract queue $Q' = op(Q[k])$.

### 3.1 Safety

Safety means that the unrolled queue corresponds to some abstract queue and that all operations change the corresponding abstract queue consistently.

**Theorem 1** (Safety). At all times $t$, an unrolled queue is in a valid state $S$, consistent with some abstract queue $Q$. All unrolled queue operations are consistent with the semantics of the abstract queue $Q$.

Trivially, if the state $S$ is valid, then the unrolled queue is also consistent with some abstract queue $Q$. We prove the theorem using induction. Initially, when the unrolled queue is empty, all the invariants hold: the unrolled queue is valid and consistent. The induction hypothesis is that the unrolled queue is valid and consistent at some time $t$. Implicitly using induction hypothesis, we show that they continue to hold.

**Lemma 1.** INV 1 $-$ head $\neq$ null

The only assignment instruction on head occurs in the dequeue operation, line 62. This instruction only changes its value to the next node, atomically. The next node could not be null because if there is only one node in the unrolled list, head = tail and the dequeue operation does not complete. The node head used to point to is considered as deleted.

**Lemma 2.** Once an array slot is set to a valid non null values, it is never set to any other value than DELETED

Array slot assignments are happen only at l. 16, l. 57 and l. 63. The atomic assignment at l. 16 is only successful when the corresponding slot equals null (which obviously is not the case here). The assignment operation at l. 57 only sets values to DELETED. The assignment at l. 63 affects only sets values to DELETED.
Lemma 3. For each node, $\text{elements}[i] = \text{DELETED}, \forall 0 \leq i < \text{deleteHint}$

$\text{deleteHint}$ is only modified in the $\text{dequeue}$ operation, at l. 57. The $\text{deleteHint}$ value is only increased to the value $i + 1$ after checking that elements from index $\text{deleteHint}$ to index $i + 1$ are set to $\text{DELETED}$. The CAS operation at l. 57 will only succeed if $\text{elements}[i]$ has been set to $\text{DELETED}$ by the current thread. Therefore, if array slots that have been set to $\text{DELETED}$ are never set to any other values than $\text{DELETED}$, the lemma holds. Which is just what lemma 2 guarantee us.

Lemma 4. INV 2 — $\text{null} \notin \text{head.elements}$

The initial node $\text{head}$ does not contain null values. $\text{head}$ is only advanced to its next node at l. 62. Using lemmas 2 and 3, and considering the loop at l. 48 - 50, we can say that all values, of this next node, prior to $\text{NODE}\_\text{SIZE} - 1$ have been checked to be set to $\text{DELETED}$ when CASing the head. $\text{elements}[\text{NODE}\_\text{SIZE} - 1] \neq \text{null}$ otherwise the if at l. 52 would have succeeded. The subsequent delete operation at l. 63 also sets the array slot to a non null value. Therefore, the lemma holds.

Lemma 5. INV 3 — $\text{tail} \notin \text{reachables(\text{head})}$

$\text{tail}$ always points to a node, reachable from $\text{head}$, in the unrolled list, because: in the $\text{dequeue}$ operation, $\text{head}$ never advances without checking that $\text{tail}$ should be set to the next index in the unrolled list. So $\text{tail}$ never references a deleted node. Also, $\text{tail}$ is never set to $\text{null}$ as l. 38-39 ensures that the next node is never set to $\text{null}$. In the $\text{enqueue}$ operation, $\text{tail}$ is CASed to a node reachable from the unrolled list (as the insertion of a new node at l. 22 has succeeded), or the the next non null node in the unrolled list (l. 28).

Lemma 6. INV 4 — $\text{node} \notin \text{reachables(\text{node.next})}$

Only new nodes are atomically added to the unrolled list (l. 21-22), so an existing node can not be added twice.

Lemma 7. For each node, $\text{elements}[i] \neq \text{null}, \forall 0 \leq i < \text{addHint}$

Using lemma 2, and considering the loop at l. 11-13, we can say that all slots prior to index $i$ are set to a non null value. When $\text{addHint}$ is increased at l. 17, an elements has successfully been enqueued at index $i$, therefore all slots prior index $i + 1$ are set to non null values.

Lemma 8. Enqueue operation is consistent: it only adds an element at the end of the unrolled queue

Elements are only added to a node that has a $\text{null} \text{next}$ reference (first condition of enqueue). The only such node is the last node of the unrolled queue (as the $\text{next}$ reference of nodes is never set to $\text{null}$). Using lemma 7, considering the loop at l. 11-13 and using lemma2, we can conclude that when enqueuing an element at l. 16 and when adding a new node to the unrolled list at l. 22, all previous slots are known to be non null. Therefore, in the $\text{elements}$
array, an element is only added at the first non null slot value. When no null slot is available (l. 20), a new node is created and is added atomically to the last node of the unrolled list. Using the induction hypothesis, and the now proven fact that an element is only added to the first non null reference, we showed that enqueue is consistent.

**Lemma 9.** Dequeue operation is consistent: it only removes an element from the beginning of the unrolled queue

Elements are only deleted from head.next (delete CAS only occurs on nh, which points to head.next). Using lemma 3, considering the loop l. 48-50 and using lemma 2, we can conclude that when dequeuing an element at l. 57 and when CASing the head at l. 62, all previous array slots are set to \textit{DELETED}. If the dequeued element is at the last position of the array, head is atomically swapped to the next node in the list. This next node is considered as deleted, and as such, the element has been dequeued.

**Corollary.** INV 5 — (Essentially $\text{elems(head.next)} = \text{DELETED}^d \cdot E^e \cdot \text{null}^n$)

Using lemmas 8 and 9 and using the induction hypothesis, INV 5 immediately falls.

The invariants always holds and both enqueue and dequeue operations are consistent. Safety is guaranteed.

### 3.2 Linearizability

An operation is linearizable if any external observer can only observe the operation as if it took place instantaneously at some point between its invocation and completion.

In our case, our operations are linearizable because there is a unique point during each operation at which it is considered to "take effect". An enqueue takes effect when:

- the element is CASed into the element array of the last node \textbf{if} a null slot is available in the array
- a newly allocated node, holding a reference to the enqueued value, is linked to the last node in the unrolled list \textbf{if} no non null slots are available in the array.

A dequeue operation takes effect when:

- the element is replaced using a CAS by a \textit{DELETED} flag \textbf{if} the element was located before the last slot of the elements array
- head swings to the next node in the list \textbf{if} the element was located at the last slot of the elements array.
3.3 Lock-freedom

Lock-freedom means that if some number of threads execute operations concurrently, then after a finite number of steps some operation must complete.

An enqueue operation loops only if the condition in line 9, the compare and swap at line 16, xor the compare and swap at line 22 fails.

A dequeue operation loops only if the condition at line 38 holds (and that the list contains more than one node), xor the CAS at line 57, or the CAS at line 62, or both conditions in lines 56 and 61 fail.

We show that the algorithm is lock-free by showing that a process loops beyond a finite number of times only if another process completes an operation on the queue.
Pseudocode 3 Dequeue operation

```java
def dequeue() = {
    loop {
        val h = head
        val nh = h.next
        val t = tail

        if (h == t) {
            if (nh == null) {
                return EMPTY
            }
            CAS(tail, t, nh)
        } else {
            var i = nh.deleteHint
            var v = null

            while (i < NODE_SIZE and v = nh.elements[i]; v == DELETED) {
                i += 1
            }

            if (v == null) {
                return EMPTY
            }

            if (i < NODE_SIZE - 1) {
                if (CAS(nh.elements, i, v, DELETED)) {
                    nh.deleteHint = i + 1
                    return v
                }
            } else if (i == NODE_SIZE - 1) {
                if (CAS(head, h, nh)) {
                    nh.elements[NODE_SIZE - 1] = DELETED
                    return v
                }
            }
        }
    }
}
```